# Design of 8-bit Microcontroller in Verilog

Chunduri Sai Abhishek[1] and Kanta D S Vijaya Raghavendra[2]

[1,2]*Indian Institute of Space Science and Technology, Trivandrum*
[1]*chunduri.sc18b114@ug.iist.ac.in*
[2]*kanta.sc18b131@ug.iist.ac.in*

July 19, 2020

# I. Introduction

RISC (Reduced Instruction Set Computer) is a design concept aimed at reducing the complexity of the instruction set, which in turn reduces the amount of space, cycle time, cost and other parameters considered during the design implementation. The advent of FPGA has allowed the implementation of the complex logical systems on FPGA. The aim of this project is to design the micro controller based on 8 bit RISC architecture using Verilog and implement it on FPGA Spartan 3E device.It takes into consideration very simple instruction set. The blocks in this micro controller are ALU, accumulator, Adders, MUX, registers and memories. It is **non-pipelined** and follows a **Harvard architecture** type memory (i.e. separate memories for program and data instructions).

# II. System being designed

The following diagram is the architecture of the microcontroller. The datapath is shown as black arrows, and control signals are red arrows.

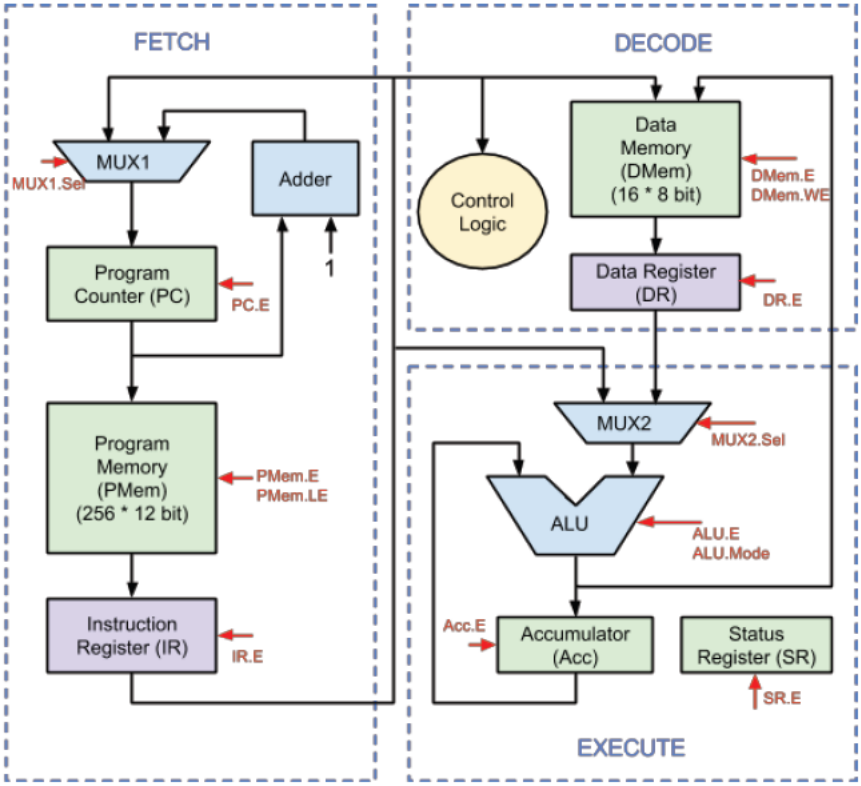



Figure 1: Block diagram representation of various modules involved

The major building blocks of the architecture are Program counter, program memory, data memory, accumulator, status register Instruction register and data register .These blocks are supported by the functional units ALU, MUX1, MUX2, Adder. Control Logic is used to denote all control signals in the system.

The non-pipelined structure uses 3 clock cycles to complete each stage , the stages are:

- **LOAD** (initial state): load program to program memory, which takes 1 cycle per instruction loaded.
- **FETCH** (first cycle): fetch current instruction from program memory.
- **DECODE** (second cycle): decode instruction to generate control logic, read data memory for operand.
- **EXECUTE** (of the third cycle): execute the instructions.

# III. Description of I/O ports in each module

## III.I Program memory

The microcontroller has a 256 x 12 bits of program memory that stores program instructions

- **Enable port** (1 bit, input, denoted as PMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out, otherwise, nothing is read out.
- **Address port** (8 bit, input, denoted as PMem.Addr): specify which instruction entry is read out.
- **Instruction port** (12 bit, output, denoted as PMem.I): the instruction entry that is read out, connected to IR.
- **Load enable port** (1 bit, input, denoted as PMem.LE): enable the load, i.e. if it is 1, then the entry specified by the address port will be load with the value specified by the load instruction input port and the instruction port is supplied with the same value; otherwise, the entry specified by the address port will be read out from the instruction port, and the value of instruction load port is ignored.
- **Load address port** (8 bit, input, denoted as PMem.LA): specify which instruction entry is loaded.
- **Load instruction port** (12 bit, input, denoted as PMem.LI): the instruction that is loaded.

## III.II Registers

These functional units are used for different functionalities.

- **Program Counter** (8 bit, denoted as PC): contains the index of current executing instruction.
- **Accumulator** (8 bit, denoted as Acc): holds result and 1 operand of the arithmetic or logic calculation.
- **Status Register** (4 bit, denoted as SR): holds 4 status bit, i.e.

  Z (zero flag, SR[3]): 1 if result is zero, 0 otherwise.

  C (carry flag, SR[2]): 1 if carry is generated, 0 otherwise.

  S (sign flag, SR[1]): 1 if result is negative (as two's complement), 0 otherwise.

  O (overflow flag, SR[0]): 1 if result generates overflow, 0 otherwise.

- **Instruction Register** (12 bit, denoted as IR): contains the current executing instruction.
- **Data Register** (8 bit, denoted as DR): contains the operand read from data memory.

## III.III Data Memory

The microcontroller has a 16 x 8 bits data memory, denoted as DMem.

- **Enable port** (1 bit, input, denoted as DMem.E): enable the device, i.e. if it is 1 then the entry specified by the address port will be read out or written in otherwise nothing is read out or written in.
- **Write enable port**(1 bit, input, denoted as DMem.WE): enable the write, i.e. if it is 1, then the entry specified by the address port will be written with the value specified by the data input port and the data output port is supplied with the same value; otherwise, the entry specified by the address port will be read out data output port, and value of data input port is ignored.
- **Address port** (4 bit, input, denoted as DMem.Addr): specify which data entry is read out, connected to IR[3:0].
- **Data input port** (8 bit, input, denoted as DMem.DI): the value that is written in, connected to ALU.Out.
- **Data output port** (8 bit, output, denoted as DMem.DO): the data entry that is read out, connected to MUX2.

## III.IV PC Adder

PC adder is used to add PC by 1,that is to move to the next instruction. This component is purely combinational. It has the following port.

- **Adder input port** (8 bit, input, denoted as Adder.In): connected to PC.
- **Adder output port** (8 bit, output, denoted as Adder.Out): connected to MUX1.In2.

## III.V ALU

ALU is used to do the actual computation for the current instruction. This component is pure combinational. It has the following port.

- **ALU operand 1 port** (8 bit, input, denoted as ALU.Operand1): connected to Acc.

- **ALU operand 2 port** (8 bit, input, denoted as ALU.Operand2): connected to MUX2.Out.
- **ALU enable port** (1 bit, input, denoted as ALU.E): connected to control logic.
- **ALU mode port** (4 bit, input, denoted as ALU.Mode): connected to control logic.
- **Current flags port** (4 bit, input, denoted as ALU.CFlags): connected to SR. ALU output port (8 bit, output, denoted as ALU.Out): connected to DMem.DI. ALU flags port (4 bit, output, denoted as ALU.Flags): the Z (zero), C (carry), S (sign), O (overflow) bits, from MSB to LSB, connected to status register.

## III.VI Control unit

Control signal is derived from the current state and current instruction. The control logic component is purely combinational. There are in total 12 control signals.The signals are listed as

- **PC.E**: enable port of program counter (PC)
- **Acc.E**: enable port of accumulator (Acc)
- **SR.E**: enable port of status register (SR)
- **IR.E**: enable port of instruction register (IR)
- **DR.E**: enable port of data register (DR)
- **PMem.E**: enable port of program memory (PMem)
- **DMem.E**: enable port of data memory (DMem)
- **DMem.WE**: write enable port of data memory (DMem)
- **ALU.E**: enable port of ALU
- **ALU.Mode**: mode selection port of ALU
- **MUX1.Sel**: selection port of MUX1
- **MUX2.Sel**: selection port of MUX2

## III.VII MUX

There are two different sets of MUXs used
**MUX1:** It is used to choose the source for updating PC
**MUX2:** It is used to choose the source for operand 2 of ALU.
The ports used are as follows:

- **MUX1 input 1 por**t (8 bit, input, denoted as MUX1.In1): connected to IR [7:0].
- **MUX1 input 2 port** (8 bit, input, denoted as MUX1.In2): connected to Adder.Out.
- **MUX1 selection port** (1 bit, input, denoted as MUX1.Sel): connected to control logic.
- **MUX1 output port** (8 bit, output, denoted as MUX1.Out): connected to PC.
- **MUX2 input 1 port** (8 bit, input, denoted as MUX2.In1): connected to IR [7:0].
- **MUX2 input 2 port** (8 bit, input, denoted as MUX2.In2): connected to DR.
- **MUX2 selection port** (1 bit, input, denoted as MUX2.Sel): connected to control logic.
- **MUX2 output port** (8 bit, output, denoted as MUX2.Out): connected to ALU.Operand2.

# IV. Verilog Code Modules

## IV.I ALU Module

```verilog
`timescale 1ns / 1ps


module ALU(
input [7:0] Operand1,Operand2,
input E,
input [3:0] Mode,
input [3:0] CFlags,
output [7:0] Out,
output [3:0] Flags

//  4 Flag bits  are  Z (zero),
// C (carry), S (sign),O (overflow)
// in order from from MSB to LSB
    );

wire Z,S,O;
reg CarryOut;
reg [7:0] Out_ALU;

always @(*)
begin

case(Mode)

// Addition Mode//

4'b0000: {CarryOut,Out_ALU} = Operand1 + Operand2;

// Subtraction Mode //

4'b0001: begin Out_ALU = Operand1 - Operand2;
CarryOut = !Out_ALU[7];
end

// Move value of accumilator to a memory //

4'b0010: Out_ALU = Operand1;

// Move value of memory entry to accumilator and Moving
    immediate number to accumilator//

4'b0011: Out_ALU = Operand2;

// Logic Gate Operations between memory entries and accumilator
    //
// ( all are bitwise operations ) //

4'b0100: Out_ALU = Operand1 & Operand2;   //  AND Gate  //
4'b0101: Out_ALU = Operand1 | Operand2;  // OR Gate    //
```

```
4'b0110: Out_ALU = Operand1 ^ Operand2; // XOR Gate    //



// Subtract Memory entry by accumilator //

4'b0111: begin
Out_ALU = Operand2 - Operand1;
CarryOut = !Out_ALU[7];
end


// ***************** Shift Micro-operations ************//


// Left Shift (Circular) //

4'b1010: Out_ALU = (Operand2 << Operand1[2:0]) | ( Operand2 >>
    Operand1[2:0]);

// Right Shift (Circular) //

4'b1011: Out_ALU = (Operand2 >> Operand1[2:0]) | ( Operand2 <<
    Operand1[2:0]);

// Logical Left Shift //

4'b1100: Out_ALU = Operand2 << Operand1[2:0];

// Logical Right Shift //

4'b1101: Out_ALU = Operand2 >> Operand1[2:0];

// Aruthmetic Shift //

4'b1110: Out_ALU = Operand2 >>> Operand1[2:0];


// ******************************************************//


// 2's complement generation //

4'b1111: begin
Out_ALU = 8'h0 - Operand2;
CarryOut = !Out_ALU[7];
end

//******************************************************//

default: Out_ALU = Operand2;
endcase
end


```

```
// ************* Assigning values to flags ************** //

assign O = Out_ALU[7] ^ Out_ALU[6];  // XOR of MSB and last one bit from MSB //
assign Z = (Out_ALU == 0)? 1'b1 : 1'b0;
assign S = Out_ALU[7];
assign Flags = {Z,CarryOut,S,O};

// ******************************************//

assign Out = Out_ALU;
endmodule
```

## IV.II Adder and MUX Modules

```
`timescale 1ns / 1ps

module adder
( input [7:0] In,
output [7:0] Out
);

assign Out = In + 1;
endmodule



module MUX1( input [7:0] In1,In2,
input Sel,
output [7:0] Out
);
assign Out = (Sel==1)? In1: In2;
endmodule
```

## IV.III Data Memory Module

```
`timescale 1ns / 1ps

module DMem(
    input clk,                  // Clock //
    input E,                  //Enable Port//
    input WE,                //Write Enable//
    input [3:0] Addr,         //Address Port //
    input [7:0] DI,           //Data In//
    output [7:0] DO           //Data Out//
    );

reg [7:0] data_mem [255:0];

always@(posedge clk)
begin

if((E==1) && (WE ==1))   // If Enable port and Write Enable ports are high, then accept data as input //
data_mem[Addr] <= DI;
```

```
end

assign DO = (E ==1)? data_mem[Addr]:0;  //If Enable port is high
    , make data available to output, else data out = zero //
endmodule
```

## IV.IV Program Memory Module

```
`timescale 1ns / 1ps

module PMem(
    input clk,                  //Clock//
    input E,                  //Enable Port//
    input [7:0] Addr,           //Address Port//
    output [11:0] I,              // Instruction Port//

    input LE,                 // Load Enable Port //
    input [7:0] LA,             // Load Address Port//
    input [11:0] LI             // Load Instruction Port//
    );

reg [11:0] Prog_Mem[255:0] ;

always @(posedge clk)
begin
if(LE == 1) begin
// When Load Enable port is high, copy instructions into Program
     Memory Register //

Prog_Mem[LA] <= LI;
end
end

assign I = (E == 1) ? Prog_Mem[Addr]: 0 ;
// When Enable is high, make instruction port to store porgram
    memory address, else it stores 'zero' //

endmodule
```

## IV.V Control Unit Module

```
`timescale 1ns / 1ps

module Control_logic(
 input[1:0] stage,     // Tells whether to LOAD or Fetch or Decode or Execute //

input [11:0] IR,       // Instruction Register //

input [3:0] SR,      //Status Register //

output reg PC_E,Acc_E,SR_E,IR_E,DR_E,PMem_E,PMem_LE,DMem_E,DMem_WE,ALU_E,MUX1_Sel,MUX2_Sel,   //Enable signals //

output reg [3:0] ALU_Mode      // ALU_Output_Mode//
    );


parameter LOAD = 2'b00,FETCH = 2'b01, DECODE = 2'b10, EXECUTE = 2'b11;


// Set all enable signals initially to 'zero' //

always @(*)
begin

PMem_LE = 0;
PC_E = 0;
Acc_E = 0;
SR_E = 0;
IR_E = 0;
DR_E = 0;
PMem_E = 0;
DMem_E = 0;
DMem_WE = 0;
ALU_E =0;
ALU_Mode = 4'd0;
MUX1_Sel = 0;
MUX2_Sel = 0;

/*************** LOAD INSTRUCTIONS *************/
if(stage== LOAD )
begin
PMem_LE = 1;
PMem_E = 1;
end

/*********** FETCH INSTRUCTIONS ****************/

else if(stage== FETCH ) begin
IR_E = 1;
PMem_E = 1;
end
```

```

/*************** DECODE INSTRUCTIONS ***************/

else if(stage== DECODE )
begin

// IF IR MSB bits are '001' then enable data registers and data memory //

if( IR[11:9] == 3'b001)
begin
DR_E = 1;
DMem_E = 1;
end

else
begin
DR_E = 0;
DMem_E = 0;
end

end

/****************** EXECUTE INSTRUCTIONS ****************/

else if(stage== EXECUTE )
begin

if(IR[11]==1) begin // ALU I-type
PC_E = 1;
Acc_E = 1;
SR_E = 1;
ALU_E = 1;
ALU_Mode = IR[10:8];
MUX1_Sel = 1;
MUX2_Sel = 0;
end

else if(IR[10]==1) // JZ, JC,JS, JO
begin
PC_E = 1;
MUX1_Sel = SR[IR[9:8]];
end

else if(IR[9]==1)
begin
PC_E = 1;
Acc_E = IR[8];
SR_E = 1;
DMem_E = !IR[8];
DMem_WE = !IR[8];
ALU_E = 1;
ALU_Mode = IR[7:4];
MUX1_Sel = 1;
MUX2_Sel = 1;
```

```
end

else if(IR[8]==0)
begin
PC_E = 1;
MUX1_Sel = 1;
end

else
begin
PC_E = 1;
MUX1_Sel = 0;
end
end

end
endmodule
```

## IV.VI Microcontroller Top Module

```
`timescale 1ns / 1ps



module MicroController(input clk,rst
    );
  parameter LOAD = 2'b00,FETCH = 2'b01, DECODE = 2'b10, EXECUTE
    = 2'b11;
  reg [1:0] current_state,next_state;
  reg [11:0] program_mem[9:0];
  reg load_done;
  reg[7:0] load_addr;
  wire[11:0] load_instr;
  reg[7:0] PC, DR, Acc;
  reg [11:0] IR;
  reg [3:0] SR;
  wire PC_E,Acc_E,SR_E,DR_E,IR_E;
 reg PC_clr,Acc_clr,SR_clr,DR_clr,IR_clr;
 wire [7:0] PC_updated,DR_updated;
 wire[11:0] IR_updated;
 wire[3:0] SR_updated;
 wire PMem_E,DMem_E,DMem_WE,ALU_E,PMem_LE,MUX1_Sel,MUX2_Sel;
 wire [3:0] ALU_Mode;
 wire [7:0] Adder_Out;
 wire [7:0] ALU_Out,ALU_Oper2;

 // LOAD instruction memory
 initial
 begin
 $readmemb("program_mem.dat", program_mem,0,9);
 end

 // ALU
 ALU ALU_unit( .Operand1(Acc),
       .Operand2(ALU_Oper2),
```

```
    .E(ALU_E),
    .Mode(ALU_Mode),
    .CFlags(SR),
        .Out(ALU_Out),
        .Flags(SR_updated) // the Z (zero), C (carry), S (sign),O (overflow) bits, from MSB to LSB, connected to status register
  );

 // MUX2
 MUX1 MUX2_unit( .In2(IR[7:0]) ,.In1(DR),
  .Sel(MUX2_Sel),
  .Out(ALU_Oper2)
     );

 // Data Memory
 DMem DMem_unit( .clk(clk),
  .E(DMem_E), // Enable port
  .WE(DMem_WE), // Write enable port
  .Addr(IR[3:0]), // Address port
  .DI(ALU_Out), // Data input port
  .DO(DR_updated) // Data output port
     );

 // Program memory
 PMem PMem_unit( .clk(clk),
  .E(PMem_E), // Enable port
  .Addr(PC), // Address port
  .I(IR_updated), // Instruction port
  // 3 special ports are used to load program to the memory
  .LE(PMem_LE), // Load enable port
  .LA(load_addr), // Load address port
  .LI(load_instr)//Load instruction port
    );

 // PC Adder
 adder PC_Adder_unit( .In(PC),
    .Out(Adder_Out)
     );

// MUX1
 MUX1 MUX1_unit( .In2(IR[7:0]) ,.In1(Adder_Out),
  .Sel(MUX1_Sel),
  .Out(PC_updated)
     );

 // Control logic
 Control_logic Control_Logic_Unit( .stage(current_state),
  .IR(IR),
  .SR(SR),
  .PC_E(PC_E),
  .Acc_E(Acc_E),
  .SR_E(SR_E),
  .IR_E(IR_E),
  .DR_E(DR_E),
```

```
  .PMem_E(PMem_E),
  .DMem_E(DMem_E),
  .DMem_WE(DMem_WE),
  .ALU_E(ALU_E),
  .MUX1_Sel(MUX1_Sel),
  .MUX2_Sel(MUX2_Sel),
  .PMem_LE(PMem_LE),
  .ALU_Mode(ALU_Mode)
    );


// LOAD
always @(posedge clk)
begin
if(rst==1) begin
load_addr <= 0;
load_done <= 1'b0;
end
else if(PMem_LE==1)
begin
load_addr <= load_addr + 8'd1;
if(load_addr == 8'd9)
begin
load_addr <= 8'd0;
load_done <= 1'b1;
end
else
begin
load_done <= 1'b0;
end
end
end

assign load_instr = program_mem[load_addr];
// next state
always @(posedge clk)
begin
if(rst==1)
current_state <= LOAD;
else
current_state <= next_state;
end
always @(*)
begin
PC_clr = 0;
Acc_clr = 0;
SR_clr = 0;
DR_clr = 0;
IR_clr = 0;
case(current_state)
LOAD: begin
if(load_done==1) begin
next_state = FETCH;
PC_clr = 1;
Acc_clr = 1;
```

```
SR_clr = 1;
DR_clr = 1;
IR_clr = 1;
end
else
next_state = LOAD;
end
FETCH: begin
next_state = DECODE;
end
DECODE: begin
next_state = EXECUTE;
end
EXECUTE: begin
next_state = FETCH;
end
endcase
end

// 3 programmer visible registers

always @(posedge clk)
begin
if(rst==1)
begin
PC <= 8'd0;
Acc <= 8'd0;
SR <= 4'd0;
end
else
begin
if(PC_E==1'd1)
PC <= PC_updated;
else if (PC_clr==1)
PC <= 8'd0;
if(Acc_E==1'd1)
Acc <= ALU_Out;
else if (Acc_clr==1)
Acc <= 8'd0;
if(SR_E==1'd1)
SR <= SR_updated;
else if (SR_clr==1)
SR <= 4'd0;
end
end

// 2 programmer invisible registers

always @(posedge clk)
begin
if(DR_E==1'd1)
DR <= DR_updated;
else if (DR_clr==1)
DR  <= 8'd0;
if(IR_E==1'd1)
```

```
 IR <= IR_updated;
 else if(IR_clr==1)
 IR <= 12'd0;
 end
endmodule
```

# V. Verilog Testbench

## V.I Testbench

```
module MCU_tb;


// Verilog project: Verilog code for microcontroller
// Inputs

reg clk;
reg rst;

// Instantiate the Unit Under Test (UUT)
MicroController uut (
.clk(clk),
.rst(rst)
);


initial begin
// Initialize Inputs
rst = 1;

// Wait 100 ns for global reset to finish
#100;
rst = 0;
end

initial begin
clk = 0;
forever #10 clk = ~clk;
end

endmodule
```

## V.I Test Programs [save them with name ("program_mem.dat")]

### V.I.I Test Program 1

```
0000_0000_0000
1011_0000_0001
0010_0010_0000
1011_0000_0000
0011_0011_0000
0001_0000_0101
0000_0000_0000
0000_0000_0000
0000_0000_0000
0000_0000_0000
```

### V.I.II Test Program 2

```
0000_0000_0000
1011_0000_0001
0010_0010_0000
0011_0000_0000
0010_0000_0000
0011_0001_0000
0010_0001_0000
0011_0111_0000
0010_0111_0000
0001_0000_1001
```

### V.I.III Test Program 3

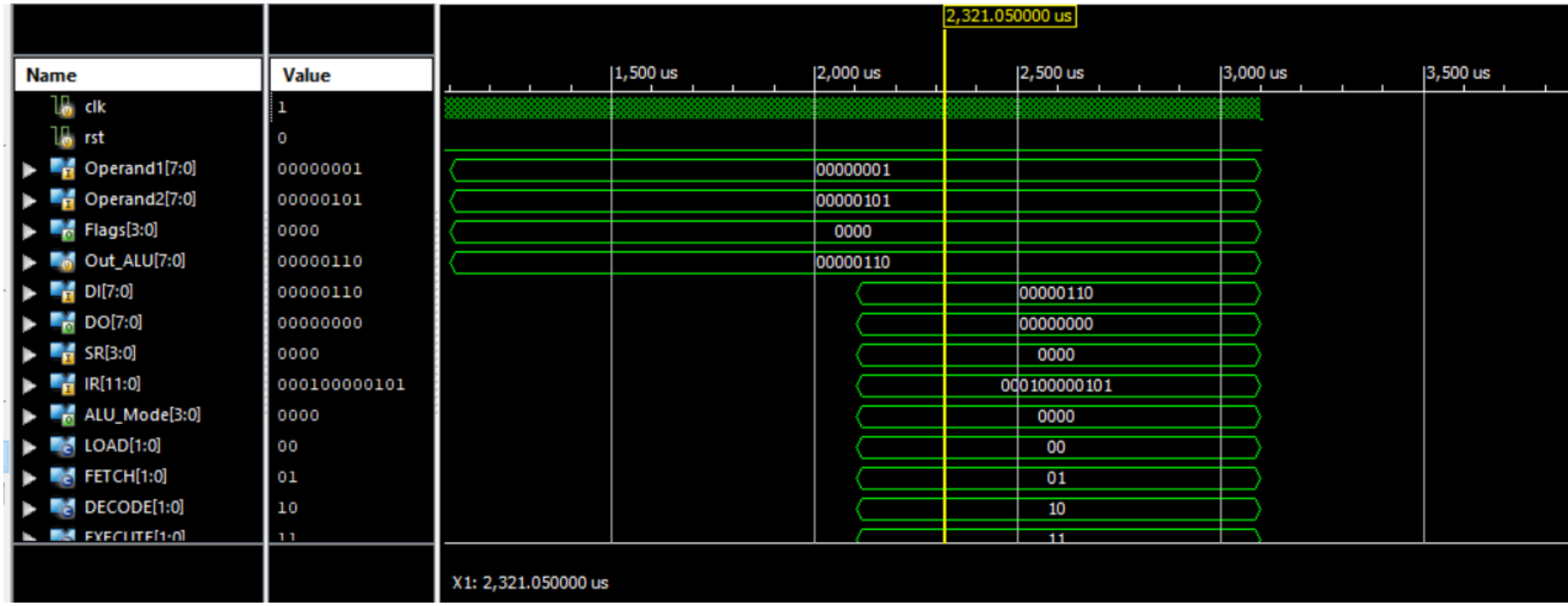
```
0000_0000_0000
1011_0000_0101
0010_0010_0000
0010_0010_0001
0010_0010_0010
1011_0000_0011
0010_0100_0000
0010_0101_0001
0010_0110_0010
0001_0000_1001
```

### V.I.IV Test Program 4

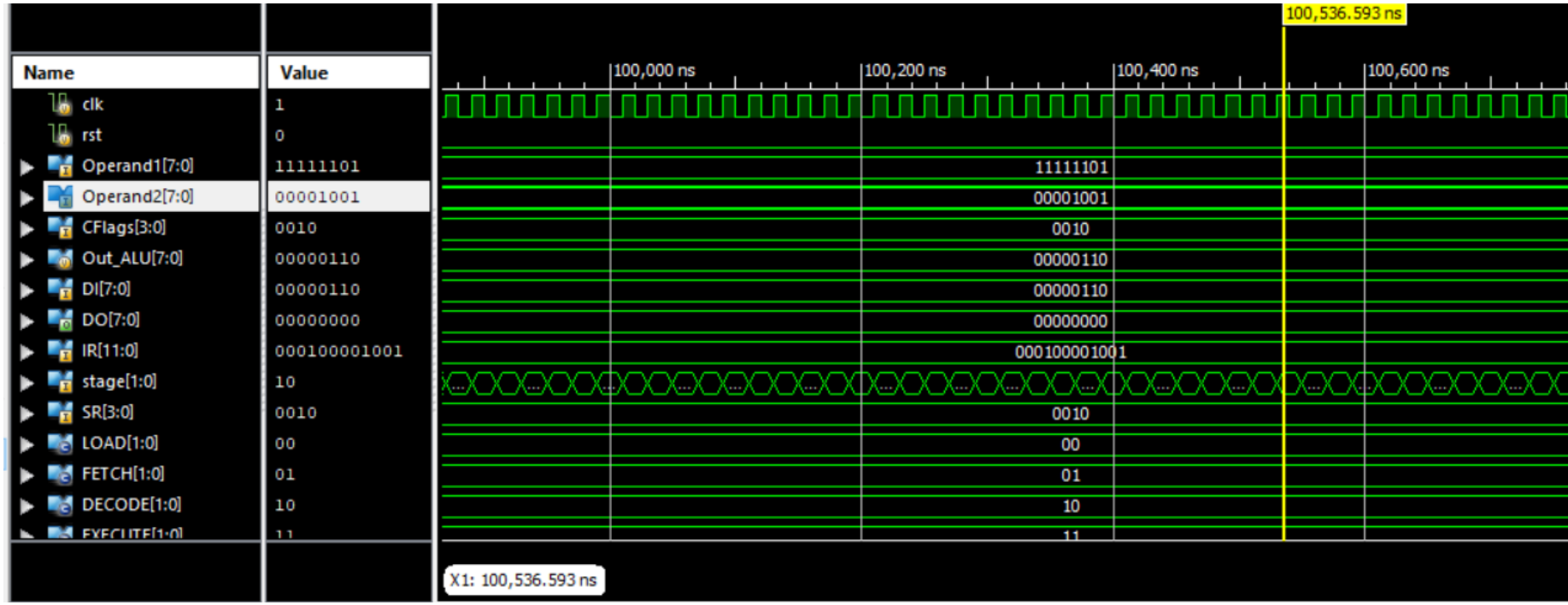
```
0000_0000_0000
1011_0000_0101
1010_0000_0000
1000_0000_0111
1001_0000_0110
1111_0000_0111
1100_0000_0011
1101_0000_0101
1110_0000_0011
0001_0000_1001
```

# VI. Simulation Results

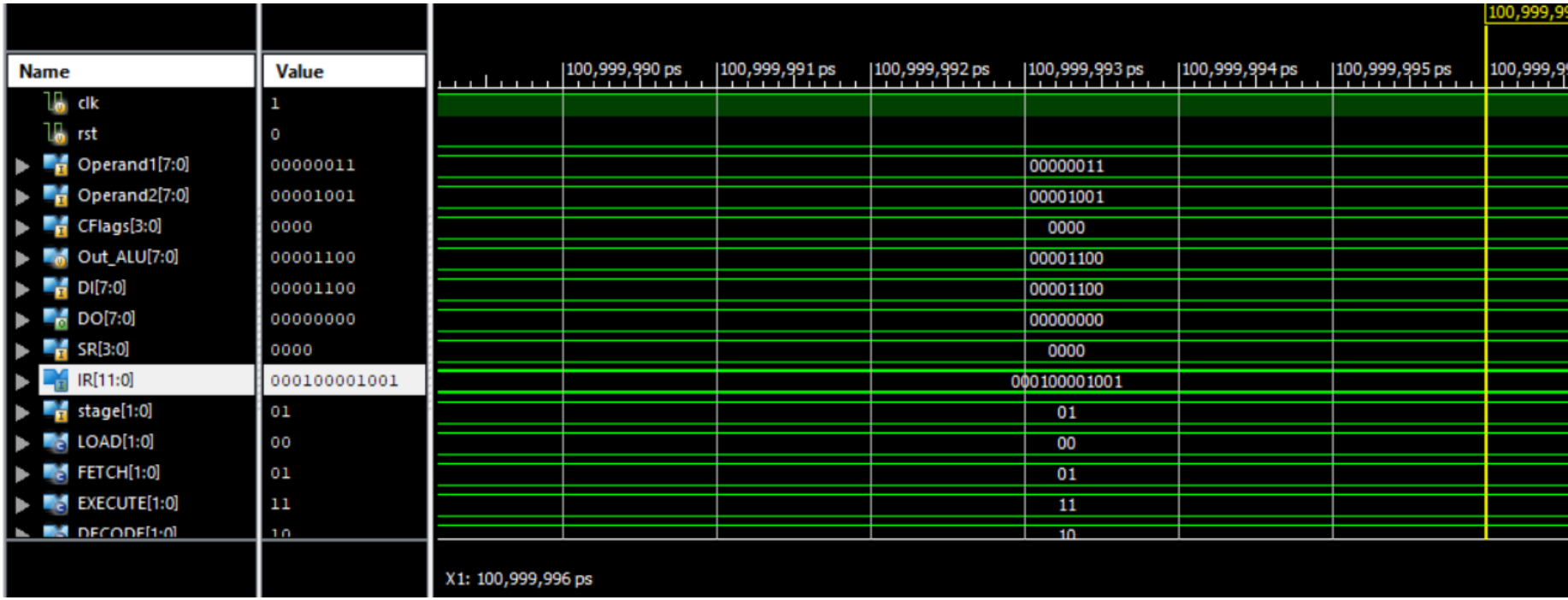
## VI.I Outputs observed with Test Program 1



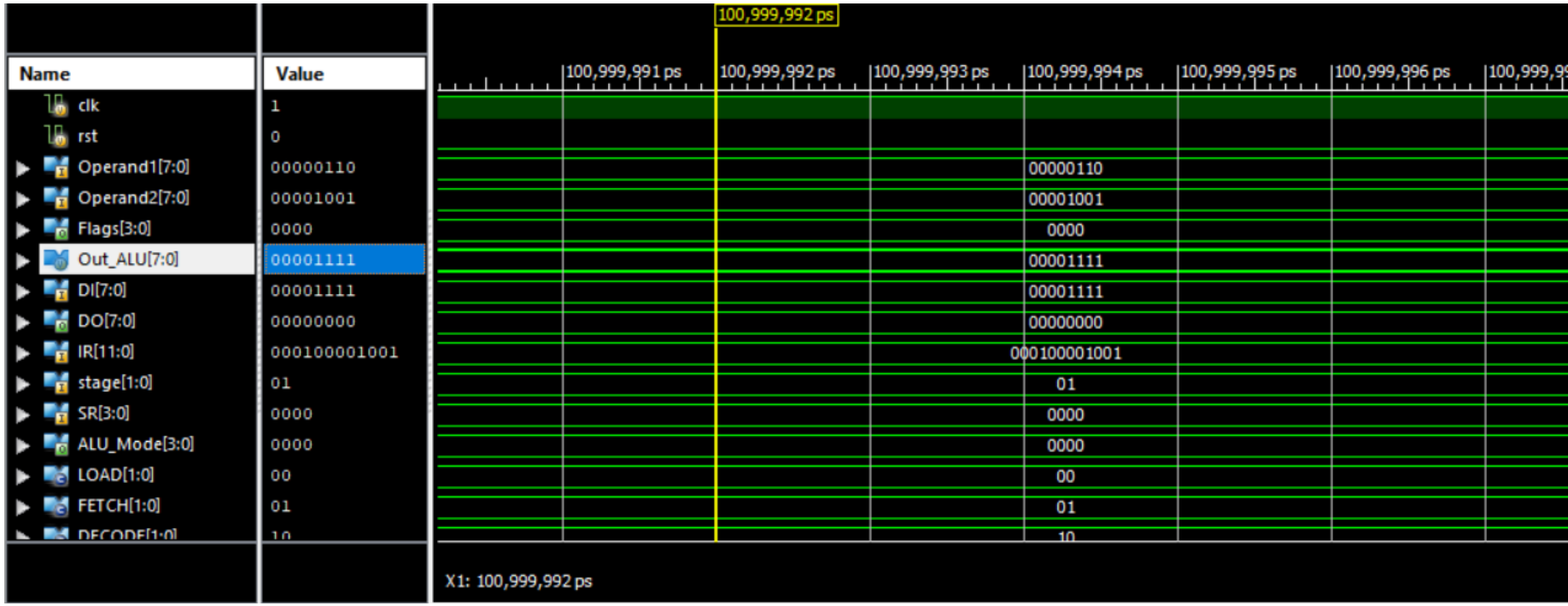
## VI.II Outputs observed with Test Program 2



## VI.III Outputs observed with Test Program 3



## VI.IV Outputs observed with Test Program 4



# VII. Contribution of Team Members

| Description | Done by |
|---|---|
| Design of ALU Module | Abhishek |
| Design of Adder and MUX Modules | Raghavendra |
| Design of Data Memory Module | Abhishek |
| Design of Program Memory Module | Raghavendra |
| Design of Control Signals Module | Abhishek |
| Design of Microcontroller Top Module | Abhishek |
| Design of Testbench/Test Programs | Raghavendra |
| Debugging | Abhishek , Raghavendra |
| Verification and Simulation | Abhishek , Raghavendra |
| Report Writing | Abhishek , Raghavendra |

# VIII. Project Files

https://drive.google.com/drive/folders/1uhJenKx8uz7m-1L169k3nM-satF63_Xd?usp=sharing